TU Clausthal
Clausthal University of Technology

Agent-based simulation for software development processes

Tobias Ahlbrecht, Jürgen Dix, Niklas Fiekas, Jens Grabowski, Verena Herbold, Daniel Honsel, Stephan Waack and Marlon Welter

IfI Technical Report Series IfI-16-02

Impressum

Publisher: Institut für Informatik, Technische Universität Clausthal
Julius-Albert Str. 4, 38678 Clausthal-Zellerfeld, Germany
Editor of the series: Jürgen Dix
Technical editor: Tobias Ahlbrecht
Contact: tobias.ahlbrecht@tu-clausthal.de
URL: http://www.in.tu-clausthal.de/forschung/technical-reports/
ISSN: 1860-8477

The IfI Review Board

Agent-based simulation for software development processes

Tobias Ahlbrecht[1] Jürgen Dix[1] Niklas Fiekas[1] Jens Grabowski[2]
Verena Herbold[2] Daniel Honsel[2] Stephan Waack[2] Marlon Welter[2]

Department of Informatics, Clausthal University of Technology
Julius-Albert-Str. 4, D-38678 Clausthal and Institue of Computer Science,
Georg-August-Universität Göttingen
Goldschmidtstrasse 7, 37077 Göttingen, Germany

Abstract

Software development is a costly process and requires serious quality control on the management level: Managing a project with more than 10 programmers over several years is a highly nontrivial task. We are building tools for helping the manager to predict the future development of the project based on certain adjustable parameters.
The main idea is to view the software process as *agent-based simulation* in a multiagent system (MAS). This approach requires to combine three different areas: (1) mining data and patterns from projects done in the past, (2) modeling the software development process in a multiagent environment (3) running the simulation on a dedicated and scalable multiagent platform.
agents, simulation, software/management processes, software evolution, mining software repositories, conditional random fields

1 Introduction

We introduce and give a bird's eye view of the *SimSe* project[1] ("Agent-based simulation models in support of monitoring the quality of software projects") that started in April 2016 and is funded by the *Simulationswissenschaftliches Zentrum (SWZ)*, a joint institution of the University of Göttingen and Clausthal University of Technology. After presenting the overall idea in Subsection 1.1, we briefly discuss related work (Subsection 1.2) and lay out the structure of this paper in Subsection 1.3.

[1] https://www.simzentrum.de/en/research-projects/agent-based-simulation-models-in-support-of-monitoring-the-quality-of-software-projects/

1.1 The very idea

The Project manager of a software project is interested in *minimizing* the number of bugs, the overall costs and at the same time *maximizing* the quality of maintenance. In order to do so, she needs answers to the following questions: (1) Where are *error-prone* parts of the code? (2) Where are candidates for *refactoring* (to improve maintenance) in the code? (3) What is the *expected effort (costs)* to achieve better results?

This leads to the following rough idea: *Simulate alternative evolutions of the project by modifying certain parameters.* The simulations can then be investigated (the quality of the software must be automatically assessed) and used to find out about suitable settings of the parameters. The resulting feedback loop is visualized in Figure 1. The problem is of course to *(1) choose the right parameters*, and, *(2) make the simulation as realistic as possible*.

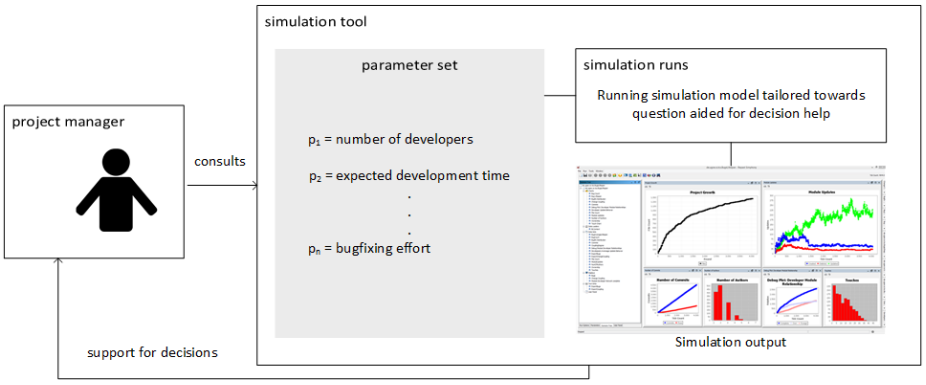


Figure 1: Feedback-cycle for project managers [14].

Where do the parameters come from? Fortunately, there is plenty of data available from many open source projects: *Firefox*[2], *MySQL*[3], ...! A prominent example popular among researches is the tera-Promise repository [18]. How to extract information from this data is considered in Section 2.

How can we simulate the evolution of software with agents? The idea is quite simple: we view *software artifacts* as *passive* agents, and *developers* (programmers) as *active* agents. Active agents *generate, extend, correct and refactor* software artifacts through *commit* actions. An illustration is given in Figure 2.

In our simulation model, elaborated in Section 3, we are simulating several important parameters, which are obtained by data mining *based on commits*

[2] `https://www.mozilla.org/en-US/firefox/`
[3] `https://www.mysql.com/`

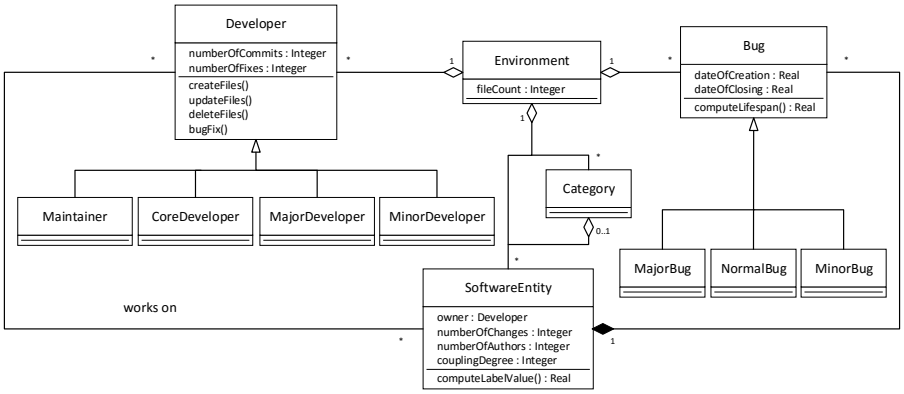


Figure 2: Agent-based simulation model for software evolution [11].

to the repository (changes of the source code):

1. the *effect and costs of refactoring*,
2. the (change in the) *behavior* of developers,
3. *communication* between developers, and
4. *goal-orientedness* and *improved experience* of developers.

1.2 Related work

Only few approaches exist in the area of monitoring software quality with simulation methods. An agent-based simulation model for software evolution was presented by Smith and Fernández-Ramil [22]. They can reproduce different facets of software evolution, e.g., the number of complex entities, the number of touches, and distinct patterns for system growth, but almost all of them need different parameter sets. The model we proposed in [11], has the following differences to this one: First, our model is not grid-based and agents do not perform a random walk. In our work, all instantiated agents live in one environment and relationships are represented as graphs. Secondly, our simulation model requires only parameters for effort and size to simulate projects that have similar growth trends.

Another interesting study is presented by Spasic and Onggo [23]. The work is aimed to support project managers in their planing by simulating possible future software processes. It is not an entirely new approach to use simulation in this context, but for a long time it was dominated by discrete event

simulation and system dynamics (because agent based simulation is a relatively new technique). The authors use data from a software department in an industrial context to estimate the simulation parameters. This work differs from other studies in that a maturity model is given (the capability maturity model integration, CMMI[4]). During the creation of the agent-based model the amount of the existing software components and the number of available developers is considered based on the design and the development phase. Then, the developers are assigned to certain (multiple) components. The components switch between different states. Finally, the model is validated by comparing the empirical project duration of different projects with the simulated results.

For the prediction of software quality in general there are many approaches in the literature. The approach in [3] is based on different software graphs. They analyzed their impact on defect-proneness and maintainability. In particular, they consider source-code based graphs and developer collaboration graphs. In our work, we also describe relations between software entities and between developers. The authors of [3] include more graphs concerning the structure of the software, e.g., call graphs, which we also plan to do. Furthermore, they compute graph metrics and correlations between these metrics and the overall software quality.

1.3 Structure of the paper

We have presented the basic structure of *SimSe* in Subsection 1.1. In order to deal with the right parameters describing the evolution of software projects, we need to determine the right patterns and mine information about the behavior of programmers based on available information. Therefore we need powerful methods from data mining. These will be investigated in Section 2.

The main part of this paper is Section 3, where we introduce the necessary software engineering constructs that we model in the simulation. They enable us to define the parameters that can be adjusted by the project manager to simulate various evolutions of the project.

What we also need is a platform where we can run our simulations and, for each such simulation, we need an assessment of how good the developed code really is: Section 4 is devoted to these tasks. We conclude with Section 5.

2 Parameter mining for the simulation model

In this section we describe how to extract necessary information from open source repositories using data mining methods. In [13, 14], we presented

[4] `http://cmmiinstitute.com/`

mining methods to obtain parameters for various simulation models. These models cover different aspects of software evolution, such as the growth of a project, bug introducing rates, or the lifespan of bugs.

With the model presented in [11] we are able to simulate the quality trend of software projects. However, the structure of the simulated *ChangeCoupling* graph is not close enough to the mined one. Thus, we have to extend the simulation model (Section 3) which leads to additional mining effort. For this extension we require more knowledge about the developer behavior and certain source-code patterns: Both are described below.

2.1 Specialized developer behavior

To estimate the effort of developers, it is of great importance to understand their driving factors and the evolution of their work. Since developers are humans, driving factors and workload depend on several factors: motivation, interests, dedication to the project, or time constraints. For the simulation of quality assurance, a deeper understanding of different types of developers is needed. The team constellation represents a simulation parameter, which has an impact on the overall project quality. For example, less active contributors may introduce more bugs.

Developers' actions are not solely visible in their commit behavior. Given the whole history of a project, it can be hard to derive a complete picture of the behavior of developers. Also their role is an important factor for the involvement in the project. For the developer role definition, we distinguish between core developers, major developers, and minor developers. These roles are assigned considering their activity over the whole time, e.g., a core developer performs over 30%, a major developer more than 2%, and minor developers more than 1% of all commits (according to earlier mining results).

For deriving a complete picture of the behavior of developers, we look at the evolution of four metrics describing the contribution: *commits*, *bugfixes*, *mailing list posts*, and *bug comments*. In order to do so, we combine the information from the version control system, the issue tracking systems and mailing lists. We are using Hidden Markov Models (HMMs) for describing this evolution in a dynamic way. HMMs are stochastic models flexible for examining discrete time observations. The set of states in our approach is $S = \{S_1,S_2,S_3\} = \{low, medium, high\}$. For $obs_1, \ldots, obs_n$ the sequence of observations as input, a HMM can be trained describing the occurring observations. n represents the project duration in months.

Figure 3 visualizes the procedure. With the Baum-Welch algorithm [] one can determine the transition probabilities between the states and the emission probabilities representing the probability that a certain observation occurs in the current state. This results in a multivariate normal distribution.

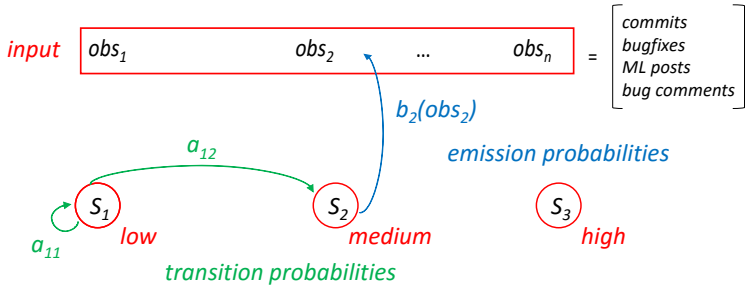


Figure 3: HMMs for developer contribution.

With the Viterbi algorithm, one can retrieve the most likely sequence of states that produced the input observations [20]. We validate this approach in our recent work [12] using six open source projects with 106 developers. There we compare individual models and general models for the different developer types. For all individual HMMs that could be trained, we calculated the misclassification rate mr indicating how many times the classified label deviates from the HMM state label. We achieve $mr = 9.8\%$ with individual models. Using general models performs about 5% worse, but can be applied even though the individual calculation is not possible, e.g., because of a small input space.

Another interesting observation occurring during our studies is the relation of the amount of developers working in a high state to the release dates of the project. Figure 4 visualizes this phenomenon. There we observed that more highly involved developers, i.e., the space between the upper and lower line decreases, corresponds to the release dates represented as the dashed lines. This matches our intuition in so far that the developers need to do more work, especially fixes and communication, before a release. However, the formalization of this relationship is in an early state of research.

2.2 Source-code change patterns

To recognize different change patterns, commits of open source projects will be analyzed. We are currently implementing a mining framework that processes commits in two main steps.

First, each commit is preliminary classified according to its size and commit message. Hattori and Lanza [10] figured out that commits can be classified in four classes concerning size. The proposed size classification is: *tiny* (1 to 5), *small* (6 to 25), *medium* (6 to 125), and *large* ($>$ 125). One further classification proposed in [10] is keyword based. For example, the words *implement*, *add*, or *new* are used if a new feature was added and the words *bug*, *issue*, and *correct* are used for bug fixes. Commits are divided into four classes: *for-*

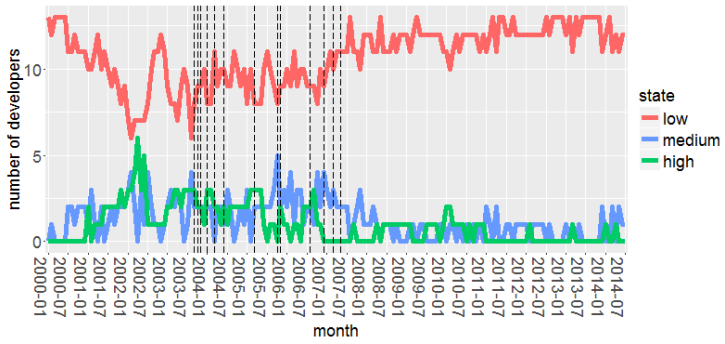


Figure 4: Amount of developers contributing in a low, medium and high state over the project duration. Dashed lines represent release dates.

ward engineering, *re-engineering*, *corrective engineering*, and *management*. Furthermore, they discovered that tiny commits are common bug fixes and that large commits occur regularly and are related to the implementation of new functionalities. Such a classification helps us to model concrete plans for different intentions of developers.

Secondly, code changes of a commit will be analyzed on abstract syntax tree level: we are using the GumTree [9] framework. It considers source code as trees and computes differences between them. These trees and differences can be visualized and exported into several formats. The tool can detect moved or renamed elements as well as deleted and inserted elements. We will recognize recurring patterns and catalog them. Even complex activities like refactorings could be revealed with this approach. With this catalog of patterns we are able to model more detailed developer plans.

Our aim is to find a large number of patterns with detailed information about what happens when one of them applies. This includes dependency changes as well as changes of software metrics like the size or the complexity of classes or methods. To get this information we have to combine this approach with previous ones.

3 Modeling the software process

As stated in [11], we propose a simulation model of software processes that predicts the quality trend of software projects. In this section, we briefly describe this model as well as its limitations and pave the way for improvements.

In the model depicted in Figure 2 we consider software entities and bugs as passive agents and developers as active ones. The developer's commit behav-

ior is responsible for the evolutionary process of the software development under simulation. Therefor, we focus on modeling the *create*, *update*, *delete*, and *bugfix* functionality of developers.

To model dependencies between entities we have chosen to use networks. This provides us with more sophisticated modeling possibilities than the grid based approach proposed in [22]. The three most important networks are described below.

- *DeveloperEntityNetwork*: This network represents the dependencies between software entities and developers. If a developer creates an entity, an edge between the developer and the new entity will be created. If a developer changes an existing entity and an edge between the developer and the entity already exists, the weight of the edge will be increased, otherwise a new edge will be created. Hence, this network provides the owner, the number of authors, and the number of changes of an entity.

- *BugEntityNetwork*: The environment creates bugs at scheduled points in time according to the mined bug introducing rate. After a bug is instantiated, an edge between the new bug and the randomly selected software entity is added to this network. The edge contains also information about whether a bug is fixed or not.

- *ChangeCouplingNetwork*: This network represents dependencies between software entities that are changed together several times. It serves as input for the automated assessment.

For a simulation run we have to parametrize the model. The required information is provided by the mining of open source repositories (see Section 2). For the concrete model instance described in [11] we used parameters of K3b [24], a Linux CD/DVD burning tool. To validate the model we have mined projects similar in size and duration to K3b and changed only few parameters of the model like the number of developers and the size. We were able to give a quality trend of these projects.

For modeling and simulation purposes we used *Repast Simphony* [19], an open source framework for agent-based simulation. This tool was well suited for modeling small to medium projects with about 100 developers and networks with up to 2000 nodes. It will not be appropriate for the large number of agents that we intend to simulate in this project.

The resulting simulation model reveals issues concerning the structure of the simulated change coupling graph and the bug fix probabilities of developers. Addressing the first issue we plan to improve the software entity selection for commits with the introduction of a more detailed software dependency graph as described in Section 3.1. For the latter issue we plan to add communication skills to the developers experience model (see Section 3.2).

3.1 Modeling developer goals and plans

One of the challenges is to model the entity selection of a commit. Without knowledge about the intention of the developer, software entities are selected mainly randomly as mentioned in [11]. This results in significant differences between a simulated and a mined, i.e., real change coupling graph.

To reduce the coincidence, we plan to use the prominent BDI [28] approach for future simulation models. In such a model, developers formulate goals based on their beliefs and build plans to reach them. One example, how the decision process of a developer leads to an action, is depicted in Figure 5.

Beliefs are the current state of the project, represented as software metrics, as well as a parameter that can be set by the manager each time the simulation runs. Thus, we can easily compare differently configured simulation runs with each other. Goals, for example, add new features, fix bugs, improve the maintainability, or reduce the complexity of the project. A developer agent selects the goal based on its beliefs. From time to time the beliefs have to be revised.

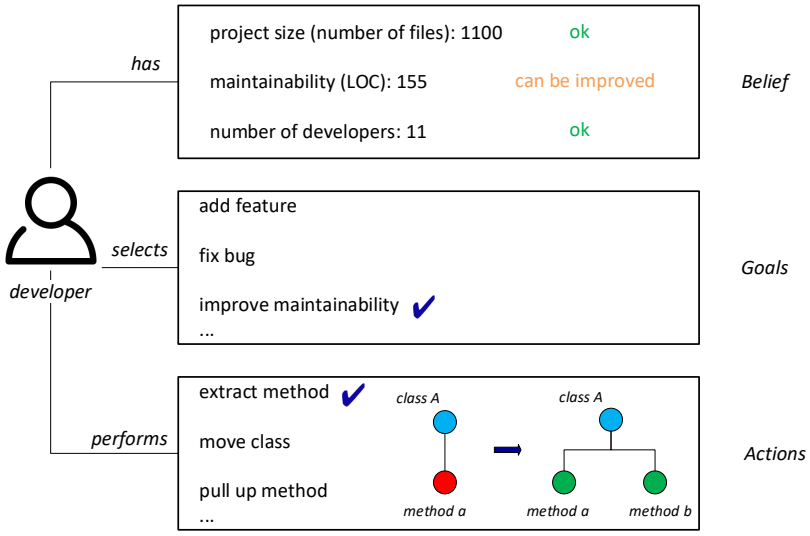


Figure 5: Example for developer's goals and plans. The developer works on a method that is hard to maintain because it has too many lines of code. To improve maintainability the developer applies the refactoring extract method that splits the method.

Plans are patterns that should, when applied to the software graph, achieve a goal. They can also be concatenated to reach a goal.

To get a realistic model we need patterns for different source code changes like refactorings, bug fixes, or additional functionality. The formulation of them requires preliminary work in terms of mining open source repositories (described in Section 2.2). Valuable information about one pattern are how software metrics like the complexity or the lines of code change, how many files will be touched and how the touched files are connected.

The change coupling graph is not well suited to apply the above introduced patterns. We need a more detailed software dependency graph. It should be detailed enough to deal with plans and goals of developers: dependencies of classes, methods, and variables need to be modeled to deal with metrics, but more abstract than the concrete syntax-tree of the project. Its size is expected to be around $10\,000$ nodes, $200\,000$ edges for medium sized projects like K3b which is about 10 times more than the change coupling graph.

We believe that this model improves the structure of the simulated change coupling graph significantly, but it also adds additional requirements on the simulation platform. Therefore, we are developing our own scalable agent platform as described in Section 4.1. With this platform we are planning to simulate large software projects like Eclipse or the Linux kernel and we shall consider software ecosystems where several software projects exchange information, resources, and entities. To achieve these goals, the platform needs to scale up in the number of agents.

3.2 Modeling communication between developers

The experience of a developer, which is an important factor of the probability to fix a bug, is closely related to the communication between the developers. In software projects communication occurs in mailing lists or issue tracking systems. How exactly this experience is influenced by communication is described in Section 2.1, where we consider mailing list posts and bug comments. To model this is a new requirement to the simulation platform. The platform proposed in Section 4.1 provides cooperation skills which allows us to model interactions between the developers. For modeling, however, not just the occurrence and the extent of communication activities, but also the intentions behind are important. It is of special interest for us how the communication relates to actions which can be retraced later in the repository. Moreover, the state of developers in communication networks based on mailing lists and ITS can be an important factor in this analysis. We are currently working on the analysis of such networks and the impact on decisions during the project.

Further investigations help us to examine whether the simulated bug fix rate of the different developer types can be improved with an extended experience model in comparison to the developer behavior model presented in

Section 2.1.

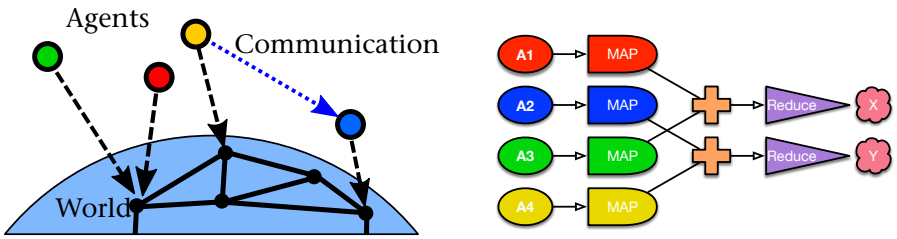
4 Implementation and Assessment

We elaborate in Subsection 4.1 on the simulation platform, in particular on the ideas to make it scalable, so that bigger projects with hundreds of thousands of agents can be simulated. We also need for each simulation a measure on how good the developed software is: Subsection 4.2 s devoted for that task.

4.1 Developing a scalable agent platform

As mentioned in the introduction, available dedicated simulation platforms (like *Repast Symphony*) or general agent languages that offer declarative tools for suitable modelling (like *Jason* ([4])) do not scale up in the number of agents and can therefore not be used for our purposes.

But *Jason*-like languages do offer interesting tools that facilitate the modelling of the simulation model for software evolution enormously (and are also reusable). In particular, *Jason*, as taken off the shelf, is extremely limited in the number of agents (only a few hundred if communication is used).



(a) Synchronized versus non-synchronized objects.

(b) The idea behind MapReduce.

Figure 6: Idea of MapReduce

In previous work ([2],[6]) we have already worked on a general agent platform, *MASeRaTi*, to deploy huge numbers of agents in the area of traffic simulation). Many techniques and design decisions will be reused. Instead of reimplementing *Jason* from scratch, we focus on a new approach, based on *MapReduce*.

The main idea, illustrated in Figures 6(a) and 6(b), is to distinguish between synchronized and non-synchronized objects and then to identify parts of the simulation that are completely independent from each other and can thus be processed in parallel. Agents that are working on the same part of the

world (green and red) or are communicating with each other (yellow, blue) need to be synchronized among them: groups doing independent work need not.

We believe that our approach is not restricted to *Jason*, but can be applied to a whole class of similar agent languages. *MapReduce* was designed to simplify parallel processing of large datasets. The main algorithm can be executed using a *MapReduce* framework like Spark[5], Hadoop[6], MR4C[7], *MapReduce*-MPI[8] or Disco[9], which automatically partition the dataset for parallel execution on a shared-nothing cluster. While we are still experimenting to find the most appropriate setting we describe and evaluate an early version of our proof of concept implementation.

The main step in our approach is to find an efficient translation from *Jason* to *MapReduce*: see [1] for a detailed discussion.

Using a particular benchmark, a counting scenario, we compare the performance of our platform running on different Python interpreters (Python 2, Python 3, and PyPy) with the performance of other platforms (*Jason* and *MASeRaTi*). For this simple benchmark from [7] all compared platforms are scaling roughly linearly as expected (even 1-10 Mio agents should work).

In contrast, *Jason* can not run the scenario for $50\,000$ agents, even without any communication[10]. We achieve the best performance with PyPy which uses Just-In-Time compilation and hotspot optimisation (see proportional speedup as number of agents increases).

Previous approaches were either restricted in the use of agent models [21] or in the expressibility of the underlying language [26]. Our platform, in contrast, supports full *Jason*-style *AgentSpeak*. Using *MapReduce* allows us to get a linear scale-up in the number of agents.

Strictly speaking, this scalability applies only to the particular benchmark used. However, we believe that it represents a situation with high throughput very well which occurs in many simulations.

4.2 Automated Assessment

Traditionally, local metrics are used to identify flaws in the source code. They are complemented by further assessment patterns like *maturity stages* [5], number of developers, activity levels (e.g. bug-fixes, mailing lists), or project out-degree [27].

Another possibility to assess a software project is based on graph structures being inherent in the project (see [25], [17], [16], and [3]). We report

[5] `http://spark.apache.org/`
[6] `http://hadoop.apache.org/`
[7] `https://github.com/google/mr4c`
[8] `http://mapreduce.sandia.gov/`
[9] `http://discoproject.org/`
[10] Note that the line for *Jason* 1.4.2 in Figure 3 ends at $10\,000$ agents.

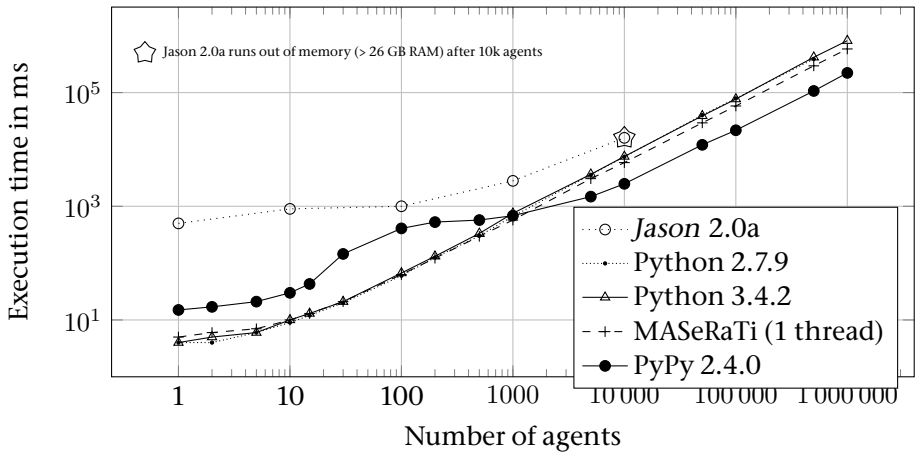


Figure 7: Execution times of the counting scenario for increasing numbers of agents

here a project under development whose graphical structure $\mathcal{G} = (\mathcal{V},\mathcal{E})$ is given by the change coupling graph. For the sake of simplifying notations, we identify the set of (current) software entities $\mathcal{V}$ with the set of its indices $\{1, 2, \ldots, m\}$.

At the current stage of the project (see [11]), the dependencies are given by change couplings. Thus, the software graph is a *change coupling graph*. Every node (software entity) $i \in \mathcal{V}$ of the simulated change coupling graph is augmented with a preliminary label *problematic* represented by $x_i = -1$ or with a preliminary label *acceptable* represented by $x_i = +1$. This classification is calculated taking software metrics of entity i and possibly of its neighbors as input. Note that the label is local in the sense that it does not depend on the labels of i's neighbors in the software graph $\mathcal{G}$. In what follows, let $\mathbf{x} := (x_1,x_2,\ldots,x_m)$ be that sequence of preliminary labels. This graph and the preliminary label sequence $\mathbf{x}$ serve as input for the automated assessment.

The automated assessment is aimed at replacing the preliminary assessment labels by final ones denoted by $y_i = +1$ (*acceptable*) or by $y_i = -1$ (*problematic*), where $i \in \mathcal{V}$. This is motivated by the fact that the overall judgment of a software entity is strongly influenced by those entities that are dependent on it. Let $\mathbf{y}$ denote the sequence $(y_1, y_2, \ldots, y_m)$ of final labels, which is the output of the automated assessment.

Taking pattern from the Ising model of statistical mechanics [15], we created a conditional random field-based model to determine the final labeling. In line with the Ising model, we introduced conformity weights h_i ($i \in \mathcal{V}$)

rewarding that preliminary label and final label of entity i coincide, and a coupling parameter J rewarding that the final labels of adjacent nodes are equal. The conditional distribution of the final labeling $\mathbf{y}$ given the preliminary labeling $\mathbf{x}$ is given by the following two equations.

$$\mathrm{p}\left(\mathbf{y} \mid \mathbf{x}\right) = \frac{1}{Z(\mathbf{x})} \exp\left(\sum_{i=1}^{m} h_i y_i x_i + \sum_{\{k,l\}\in\mathcal{E}} J y_k y_l\right), \tag{1}$$

$$Z(\mathbf{x}) := \sum_{\mathbf{y}'\in\mathcal{B}^V} \exp\left(\sum_{i=1}^{m} h_i y'_i x_i + \sum_{\{k,l\}\in\mathcal{E}} J y'_k y'_l\right). \tag{2}$$

Given the software graph $\mathcal{G}$, the weights h_i ($i \in \mathcal{V}$) and J, and the preliminary labeling $\mathbf{x}$, to calculate the final labeling $\mathbf{y}^*$, we make a *maximum posterior probability (MAP)* prediction: $\mathbf{y}^* = \mathrm{argmax}_{\mathbf{y}}\, \mathrm{p}\left(\mathbf{y} \mid \mathbf{x}\right)$. Since this problem is NP-hard, we adopted to this end a Viterbi heuristics devised by Dong *et al.* in [8]. In [11] we "semi-automatically" determined the weights h_i ($i \in \mathcal{V}$) and J such that

- dependencies between software entities influence the quality of each other, particularly in a way that a problematic software entity negatively influences those entities that depend on it.
- highly interconnected "communities" are (more or less) homogeneously labeled, whereas communities only slightly interact.

As far as assessment is concerned, here are the future challenges:

1. Devise an algorithm to determine the weights h_i ($i \in \mathcal{V}$) and J automatically according to the foregoing principle.
2. Adapt the Viterbi heuristics devised in [8] to very large software graphs.
3. Adapt known sampling algorithms using Markov Chain Monte Carlo Methods to be able to replace final labels Y_i ($i \in \mathcal{V}$) by its posterior probabilities $\mathbb{P}\left(Y_i = \text{acceptable} \mid \mathbf{x}\right)$.
4. Check the applicability of the assessment approach by analyzing and simulating open source projects.

5 Conclusions and outlook

The project reported in this paper is a continuation of two previous projects (`https://simzentrum.de/en/`

education/softwarequalitaetssicherung-mit-hilfe-von-simulationsverfahren/ and https://simzentrum.de/en/research-projects/desim/) and is scheduled for 3 years. We can therefore build on solid foundations and experiences. In order to make more precise predictions of the behavior of the developers, we need to model their plans and intentions. Therefore we have chosen *Jason* which provides language constructs for suitable modeling.

Our aim in the future is fourfold: (1) We have to find out which other constructs we need for suitable modeling, (2) how to integrate them in a *scalable* simulation platform, (3) how to mine appropriate information from open source repositories, and (4) develop an overall simulation model (as an extension of the current one) that takes all these tasks into account.

Acknowledgment

The authors thank the SWZ Clausthal-Göttingen[11] that partially funded our work (both the former projects “Simulation-based Quality Assurance for Software Systems” and “DeSim”, and the recent project “SimSe”).

References

[1] Tobias Ahlbrecht, Jürgen Dix, and Niklas Fiekas. Scalable multi-agent simulation based on mapreduce (forthcoming). Technical Report IfI-16-03, TU Clausthal, 2016.

[2] Tobias Ahlbrecht, Jürgen Dix, Niklas Fiekas, Philipp Kraus, and Jörg P. Müller. An architecture for scalable simulation of systems of cognitive agents. *International Journal of Agent-Oriented Software Engineering (forthcoming)*, 2016.

[3] Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu, and Michalis Faloutsos. Graph-based analysis and prediction for software evolution. In *Proceedings of the 34th Intern.Conf. on Softw. Eng. (ICSE)*. IEEE, 2012.

[4] Rafael H. Bordini, Jomi F. Hübner, and Michael Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. Wiley & Sons, 2007.

[5] Kevin Crowston and Barbara Scozzi. Exploring the strengths and limits of open source software engineering processes: A research agenda. In *Proceedings of the 2nd ICSE Workshop on Open Source*, 2002.

[11] https://www.simzentrum.de/en/

[6] Fabiano Dalpiaz, Jürgen Dix, and M. Birna van Riemsdijk, editors. *Engineering Multi-Agent Systems - Second International Workshop, EMAS 2014, Paris, France, May 5-6, 2014, Revised Selected Papers*, volume 8758 of *Lecture Notes in Computer Science*. Springer, 2014.

[7] Álvaro Fernández Díaz, Clara Benac Earle, and Lars-Åke Fredlund. eJason: An implementation of Jason in Erlang. In *International Workshop on Programming Multi-Agent Systems*, pages 1–16. Springer, 2012.

[8] Zhijie Dong, Keyu Wang, Truong Khanh Linh Dang, Mehmet Gültas, Marlon Welter, Torsten Wierschin, Mario Stanke, and Stephan Waack. Crf-based models of protein surfaces improve protein-protein interaction site predictions. *BMC Bioinformatics*, 15(1):1–14, 2014.

[9] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Montperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, New York, NY, USA, 2014. ACM.

[10] Lile Hattori and Michele Lanza. On the nature of commits. In *ASE Workshops*, pages 63–71. IEEE, 2008.

[11] Daniel Honsel, Verena Honsel, Marlon Welter, Jens Grabowski, and Stephan Waack. Monitoring Software Quality by Means of Simulation Methods. In *10th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2016.

[12] Verena Honsel, Steffen Herbold, and Jens Grabowski. Hidden markov models for the prediction of developer involvement dynamics and workload. In *12th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*, 2016.

[13] Verena Honsel, Daniel Honsel, and Jens Grabowski. Software process simulation based on mining software repositories. In *ICDM Workshop*, 2014.

[14] Verena Honsel, Daniel Honsel, Steffen Herbold, Jens Grabowski, and Stephan Waack. Mining software dependency networks for agent-based simulation of software evolution. In *ASE Workshop*, 2015.

[15] Ernst Ising. Beitrag zur Theorie des Ferromagnetismus. *Zeitschrift für Physik A Hadrons and Nuclei*, 1925.

[16] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. Power laws in software. *ACM Trans. Softw. Eng. Methodol.*, 18(1):2:1–2:26, October 2008.

[17] Yutao Ma, Keqing He, and Dehui Du. A qualitative method for measuring the structural complexity of software systems based on complex networks. *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, 0:257–263, 2005.

[18] Tim Menzies, Mitch Rees-Jones, Rahul Krishna, Carter Pape, and David Pryor. The tera-promise repository of empirical software engineering data, 2016.

[19] Michael J. North, Nicholson T. Collier, Jonathan Ozik, Eric R. Tatara, Charles M. Macal, Mark Bragen, and Pam Sydelko. Complex adaptive systems modeling with repast simphony. *Complex Adaptive Systems Modeling*, 2013.

[20] L. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[21] Atanas Radenski. Using MapReduce Streaming for Distributed Life Simulation on the Cloud. ECAL, 284-291(2013), 2013.

[22] Neil Smith and Juan Fernández Ramil. Agent-based simulation of open source evolution. In *Software Process Improvement and Practice*, 2006.

[23] Bojan Spasic and Bhakti S. S. Onggo. Agent-based simulation of the software development process: a case study at avl. In Oliver Rose and Adelinde M. Uhrmacher, editors, *Winter Simulation Conference*, pages 400:1–400:11. WSC, 2012.

[24] Sebastian Trueg. K3b – The CD/DVD Kreator for Linux. http://www.k3b.org/, 2011.

[25] S. Valverde and R. V. Solé. Hierarchical Small Worlds in Software Architecture. *arXiv: cond-mat/0307278*, 2003.

[26] Guozhang Wang, Marcos Antonio Vaz Salles, Benjamin Sowell, Xun Wang, Tuan Cao, Alan J. Demers, Johannes Gehrke, and Walker M. White. Behavioral simulations in mapreduce. *CoRR*, abs/1005.3773, 2010.

[27] Yu Wang. *Prediction of success in open source software development*. PhD thesis, Citeseer, 2007.

[28] Gerhard Weiss. *Multiagent Systems*. MIT Press, 2013.